

4.2 Defining Our Own Data Types, Part 1

Up to this point, all the data we've worked with in Python have been stored in objects that are instances of the built-in types that come with Python, like `ints` and `lists`. Python's built-in data types are powerful, but are not always the most intuitive way to store data. For example, we saw in 4.1 Tabular Data that we could use a list of lists to represent tabular data. One of the downsides of this approach is that when working with this data, the onus is on us to remember which list element corresponds to which component of the data.

```
>>> import datetime
>>> row = [1657, 'ET', 80, datetime.date(2011, 1, 1)]
>>> row[0] # The id
1657
>>> row[1] # The name of the civic centre
'ET'
>>> row[2] # The number of marriage licenses issued
80
>>> row[3] # The time period
datetime.date(2011, 1, 1)
```

You can imagine how error prone this might be. A simple “off by one” error for an index might retrieve a completely different data type. It also makes our code difficult to read; the reader must know what each index of the list represents. And, as more experienced programmers will tell you, readable code is crucial.¹

¹ “Any fool can write code that a computer can understand. Good programmers write code that humans can understand.” –Martin Fowler

So a row in our marriage license data set is made up of four data elements. It would be nice if, instead of indices, we could use a name that was reflective of each element. Certainly, we could use a dictionary (instead of a list) where the keys are strings. But there is a more robust option we'll learn about in this section: creating our *own* data types.

Defining a data class

You might remember from Chapter 1 that in Python, another term for data type is a **class**.²

² This is why `type(3)` evaluates to `<class 'int'>` in Python.

The built-in data types we've studied so far illustrate how rich and complex data types can be. So for creating our own data types, we will first learn about the simplest kind of data

type: a **data class**, which is a kind of class whose purpose is to bundle individual pieces of data into a single Python object.

For example, suppose we want to represent a “person” consisting of a given name, family name, age, and home address. We already know how to represent each individual piece of data: the given name, family name, and address could be strings, and the age could be a natural number. To bundle these values together, we could use a list or other built-in collection data type, but that approach would run into the issues we discussed above.

So instead, we define our own data class to create a new data type consisting of these four values. Here is the way to create a data class in Python:

```
from dataclasses import dataclass

@dataclass
class Person:
    """A custom data type that represents data for a person."""
    given_name: str
    family_name: str
    age: int
    address: str
```

Let’s unpack this definition.

1. `from dataclasses import dataclass` is a Python import statement that lets us use `dataclass` below.
2. `@dataclass` is a Python *decorator*. We’ve seen decorators before for function definitions; a decorator for a class definition works in the same way, acting as a modifier for our definition. In this case, `@dataclass` tells Python that the data type we’re defining is a data class, which we’ll explore the benefits of down below.
3. `class Person:`, signals the start of a *class definition*.³ The name of the class is `Person`.

³ This is similar to function definitions, except we use the `class` keyword instead of `def`.

The rest of the code is indented to put it inside of the class body.

4. The next line is a docstring that describes the purpose of the class.
5. Each remaining line (starting with `given_name: str`) defines a piece of data associated with the class; each piece of data is called an **instance attribute** of the class.

For each instance attribute, we write a name and a type annotation.⁴

⁴ This is similar to defining parameter names and types for functions, though of course the purposes are different.

General data class definition syntax

In general, a data class definition in Python has the following syntax:

```
@dataclass
class <ClassName>:
    """Description of data class.
    """
    <attribute1>: <type1>
    <attribute2>: <type2>
    ...
```

Using data classes

Now that we've seen how to define a data class, we now are ready to actually put it to use. To create an instance of our Person data class, we write a Python expression that calls the data class, passing in as arguments the values for each instance attribute:

```
>>> david = Person('David', 'Liu', 100, '40 St. George Street')
```

Pretty cool! That line of code creates a new Person object whose given name is 'David', family name is 'Liu', age is 100, and address is '40 St. George Street', and stores the object in the variable david. The *type* of this new value is, as we'd expect, Person:

```
>>> type(david)
<class Person>
```

If we ask Python to evaluate the Person object, we see the different pieces of data that have been bundled together:

```
>>> david
Person(given_name='David', family_name='Liu', age=100, address='40 St. George
Street')
```

But from a Person object, how do we extract the individual values we bundled together? If we were using lists, we'd simply do list indexing: david[0], david[1], etc. The syntax for Python classes improves this because we can use the names of the instance attributes together with **dot notation** to access these values:

```
>>> david.given_name
'David'
>>> david.family_name
```

```
'Liu'  
>>> david.age  
100  
>>> david.address  
'40 St. George Street'
```

This is much more readable than list indexing, and this is one of the major advantages of using data classes over lists to represent custom data in Python.

Tip: naming attributes when creating data class instances

One challenge when creating instances of our data classes is keeping track of which arguments correspond to which instance attributes. In the expression `Person('David', 'Liu', 100, '40 St. George Street')`, the order of the arguments must match the order the instance attributes are listed in the definition of the data class—and it's our responsibility to remember this order. Think about how easy it would be for us to write `Person('Liu', 'David', 100, '40 St. George Street')`, only to discover much later in our program that we accidentally switched this poor fellow's given and family names!

To solve this issue, Python enables us to create data class instances using *keyword arguments* to explicitly name which argument corresponds to which instance attribute, using the exact same format as the `Person` representation we saw above:

```
>>> david = Person(given_name='David', family_name='Liu', age=100,  
                  address='40 St. George Street')
```

Not only is this more explicit, but using keyword arguments allows us to pass the values in any order we want:

```
>>> david = Person(family_name='Liu', given_name='David', address='40 St.  
                  George Street', age=100)
```

This is a great improvement for the readability of our code when we use data classes, especially as they grow larger. One potential downside that comes with this (and in general when being more explicit) is that this requires a bit more typing, and makes our code a little longer. You can get around the first issue by using auto-completion features (e.g., in PyCharm), and for the second issue you can put the different arguments on separate lines:

```
>>> david = Person(  
...     family_name='Liu',  
...     given_name='David',  
...     address='40 St. George Street',
```

```
...     age=100
... )
```

Representing data classes in the memory model

Now that we have the ability to define our own data types, we need to decide how these data types will fit into our memory model. We'll do this by using the representation that Python displays, formatted to show each instance attribute on a new line. For example, we would represent the david variable in a memory model as follows:

Variable	Value
david	<pre>Person(family_name='Liu', given_name='David', address='40 St. George Street', age=100)</pre>